Stiftung University of Hildesheim
Marienburger Platz 22
31141 Hildesheim
Germany

Software Systems Engineering (SSE)
Institute for Computer Science
Faculty for Mathematics, Natural Science, Economics, and Computer Science

EASy-Producer

Engineering Adaptive Systems

Developers Guide

Version 1.0

20.09.2013

Version

0.1	28.08.2012	Initial version
0.2	10.09.2012	Reasoning section moved to the end of the document, prerequisite and installation added, debug flags added to section 3
0.3	04.12.2012	Preface added, Section 3.1.1 added
0.4	04.03.2013	Instantiator and reasoning section updated
0.5	01.04.2013	General corrections, e.g. spelling.
0.6	15.08.2013	Initial inclusion of VIL
1.0	20.09.2013	VIL section updated, according updates to the instantiator section

Preface

EASy-Producer is a Software Product Line Engineering tool developed by the Software Systems Engineering (SSE) group at the University of Hildesheim.

The tool is available as an Eclipse plug-in under the terms of the Eclipse Public License (EPL) Version 1.0

The SSE group hosts the following EASy-Producer update site for easy installation and updates:

http://projects.sse.uni-hildesheim.de/easy/

Table of Contents

1. Introduction

EASy-Producer[1] is a Software Product Line Engineering (SPLE) tool which facilities the most recent trends and concepts in SPLE, such as large-scale Multi-Software Product Lines (MSPL), product line hierarchies, and staged configuration and instantiation. The focus of this tool is to support these rather complex concepts in an easy-to-use way. Thus, this tool allows developing a first prototypical Software Product Line (SPL) within minutes. Further, EASy-Producer is a research prototype for demonstrating new approaches to SPLE in general and, in particular approaches for simplifying the development of SPLs developed by the Software Systems Engineering group (SSE) at the University of Hildesheim.

This live-document provides a developers guide that introduces the reader to the basic capabilities of EASy-Producer and how to develop further extensions to this tool. In Section 2, we will give guidance for the first steps with EASy-Producer. This section includes the mandatory prerequisites, the installation guide, and additional recommendations for running the tool successfully. This also provides the development environment, in which extensions for EASy-Producer will be created.

Section 3 will describe the different extension mechanisms of EASy-Producer. This includes the implementation of new instantiators, new artefact types, and new reasoners. For each of these extensions, we will briefly introduce the basic concepts and provide a step-wise example of how to create a new extension of the specific type.

[1] EASy is an abbreviation for Engineering Adaptive Systems.

2. Installation

In this section, we will describe the installation of EASy-Producer. In order to guarantee a successful installation, we will introduce a set of mandatory prerequisites. This will be part of Section 2.1 in which we will set up the environment for EASy-Producer. In Section 2.2, we will describe the installation of the tool in a step-wise manner using the Eclipse update site mechanism and the EASy-Producer update site. Finally, Section 2.3 will give some technical recommendations, while Section 2.4. introduces additional guides and specifications for EASy-Producer.

2.1. Prerequisites

EASy-Producer is developed as an **Eclipse**[2] plug-in and requires **Xtext**[3] **version 2.3.1**. Thus, in general, any Eclipse installation with Xtext version 2.3.1 is fine for installing and running EASy-Producer. However, we cannot guarantee that any combination of Eclipse and Xtext version 2.3.1 will work with EASy-Producer. Thus, we propose the following Eclipse versions as they are tested with EASy-Producer (and Xtext version 2.3.1):

- Eclipse 3.6 (Helios)
- Eclipse 3.7 (Indigo)
- Eclipse 4.0 (Juno)

We recommend using **Eclipse 3.7 (Indigo)** as this is the most exhaustively tested version of Eclipse with EASy-Producer. Download an Eclipse package from http://www.eclipse.org/downloads/.

Please note that Eclipse 4.2 does not work with Xtext 2.3.1 due to incompatible dependencies.

Further, Xtext version 2.3.1 has to be installed in the newly downloaded Eclipse instance. Typically, this is installed automatically when installing EASy-Producer due to plug-in dependencies. However, we encountered situations in which these dependencies were not automatically resolved. Thus, the EASy-Producer update site includes the required Xtext features. We will describe the complete installation in the next Section.

2.2. Installation: Step by Step

The SSE group hosts an EASy-Producer update site for easy installation and updates. Thus, the first step for installing EASy-Producer is to define a new update site in Eclipse. For this purpose, start Eclipse and open the *Install New Software* dialog by clicking *Help → Install New Software…* as shown in Figure 1:

[2] Eclipse website: www.eclipse.org/
[3] Xtext website: http://www.eclipse.org/Xtext/

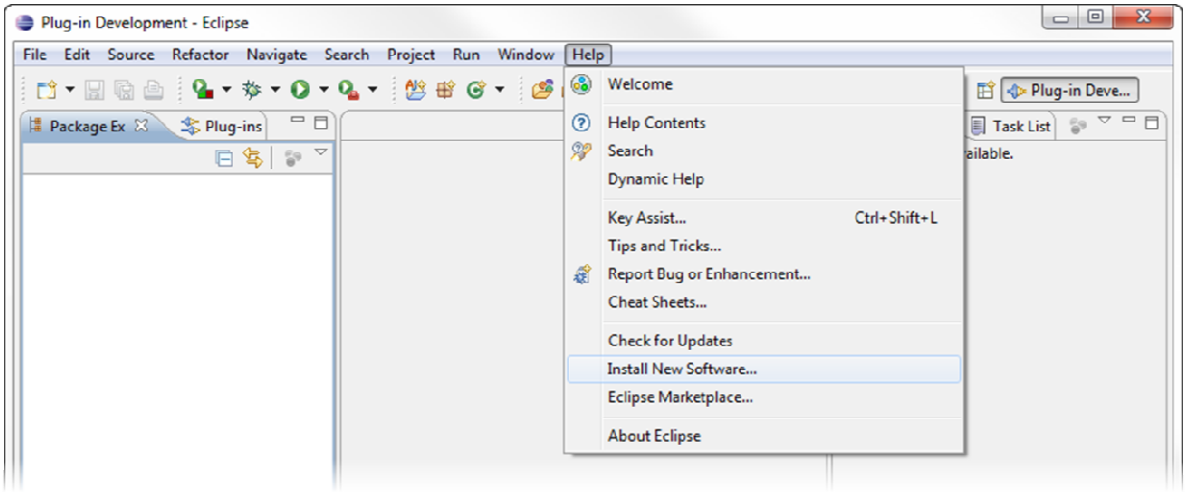

Figure 1: Open the "Install New Software" dialog

The *Install* Dialog will appear (cf. Figure 2). In this dialog, a new location for available software has to be added. Thus, click on the *Add…* button in the upper right location of the dialog.

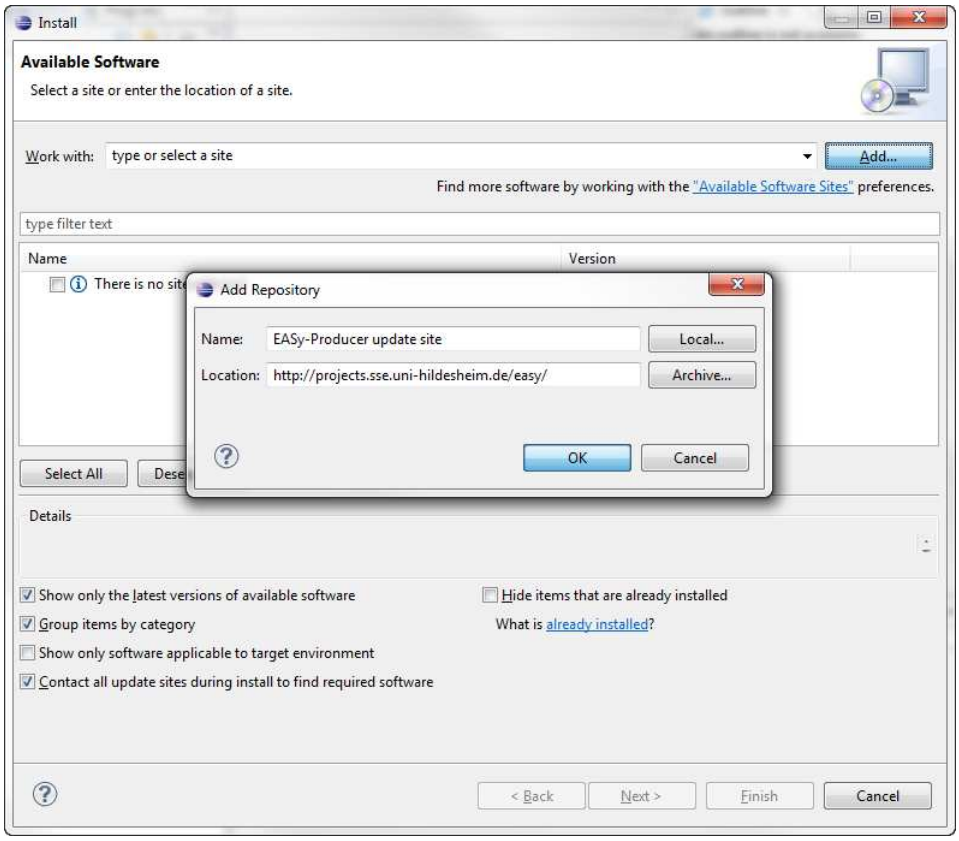

Figure 2: Add a new location for software updates

The *Add Repository* dialog requires the definition of a name for the new update site and a location as illustrated in Figure 2. The name is up to the user. For example, enter "*EASy-Producer update site*". The location is the URL of the update site:

EASy-Producer update site: http://projects.sse.uni-hildesheim.de/easy/

Finish the definition of the new update site by clicking the *OK* button of the *Add Repository* dialog.

The *Install* Dialog will now contain multiple categories. If you are installing EASy-Producer for the first time and do not know which features to select, select the *Quick Installation of EASy-Producer* category. Further, select the categories *Xtend-2.3.1* and *Xtext-2.3.1* to install the required Xtext version (if not done before). This will install all required components automatically.

For more experienced users, select the categories and features as needed and click the *Next* button. Follow the steps for installing EASy-Producer (accept the license agreement and ignore the security warning for installing software of unsigned content, etc.), and restart Eclipse as prompted.

Finally, you have successfully installed the EASy-Producer.

2.3. Technical Recommendations

In order to avoid memory problems while using EASy-Producer, we recommend increasing the memory of the Eclipse application in which EASy-Producer is executed. The memory problems are due to Xtext which requires more memory than defined in a typical Eclipse configuration.

Open the "*eclipse.ini*" file in your Eclipse directory and enter the following parameters at the end of the file:

```
-vmargs

-Xms40m

-Xmx512m

-XXMaxPermSize=128m
```

2.4. Further Guides and Specifications

EASy-Producer provides two expressive languages that support the creation of required software product line artefacts:

The **INDENICA Variability Modelling Language (IVML)** is and expressive, textual variability modelling language, which provides basic and advanced modelling capabilities for the definition of variability models. In order to define such a model based on IVML, we provide the IVML language specification. This specification is part of the EASy-Producer installation and can be found in the **Eclipse Help**.

The **Variability Implementation Language (VIL)** is a textual language for the flexible specification of the instantiation process of a software product line. This language consists (beside other parts) of the VIL build language and the VIL template language. The former language provides modelling elements for the specification of the individual build tasks of the instantiation process, while the latter language supports the definition of templates that can be applied to specific artefacts, for example, to manipulate their content, as part of the instantiation process. The corresponding VIL language specification is also part of the EASy-Producer installation and can be found in the **Eclipse Help**.

Further, EASy-Producer provides a user guide, which introduces the reader to the basic concepts and the different capabilities of the tool. The **EASy-Producer User Guide** can be found in the **Eclipse Help** as well.

The EASy-Producer user guide, the EASy-Producer developers guide, as well as the IVML and the VIL language specification are also available as PDFs on the EASy-Producer update site.

3. EASy-Producer Extensions

EASy-Producer provides an extension point mechanism to add additional functionality to the basic implementation. An extension is always implemented as an Eclipse plug-in and may provide customer-specific functionalities in terms of individual instantiators, artefact types, or reasoners. Custom instantiators may be capable of instantiating artefacts of different types or in a specific way. Artefact types will enable the definition and manipulation of specific artefacts as part of the instantiation process. A new reasoner may provide new or adapted capabilities to check, for example, whether a variability model or a specific product configuration is valid. In order to ease the development and integration of such extensions, EASy-Producer is capable of automatically searching and integrating new plug-ins through Eclipse Dynamic Services[4]. Thus, developers only have to provide the necessary information to EASy-Producer to include their desired functionalities.

In this section, we will describe how to implement extensions to the EASy-Producer tool. In Section 3.1, we will describe the implementation of a new instantiator and its integration in EASy-Producer. Section 3.2 will describe the implementation and integration of a new artefact type, while in Section 3.3, we will implement and integrate a new reasoner. Each section will provide detailed guidance from project creation and configuration to the final intragetion of the custom plug-ins in EASy-Producer.

In order to debug errors and failures during the development of EASy-Producer extensions, add the following flags to the "Run Configuration" of your Eclipse as needed (introduce the new flags with a single prefixed "-D" in the Run Configuration):

- **-debug:** This flag will print information on the variability model of EASy-Producer
- **-log:** This flag will print EASy-internal debug messages, such as errors, etc.
- **-equinox.ds.debug:** This flag will print debug messages regarding the service registration mechanism. For more details, see Section 3.1.2.
- **-equinox.ds.print:** This flag will print additional information regarding the service registration mechanism. For more details, see Section 3.1.2.

Please note that the above flags are optional. They are not a prerequisite for creating extensions for EASy-Producer but may help searching and correcting errors.

3.1. Implementing a New Instantiator

An instantiator is an external and maybe third-party tool that processes product line artefacts in its specific way. For example, the Velocity instantiator, which is shipped as a default instantiator with EASy-Producer, resolves Velocity-specific tags within Java code in accordance to a specific configuration[5]. This resolution capability allows deriving individual product variants based on the configuration values and the corresponding manipulation of Java code. However, the default instantiators of EASy-Producer may be insufficient in some situations. Further, in some situations it is the better choice to realize a proper integration, e.g., if a legacy executable is used for

[4] For more information visit: http://eclipse.org/equinox/

[5] Details on the Velocity instantiator can be found in the EASy-Producer user guide (cf. Section 2.4).

instantiation (this may be called directly from VIL) and the modified artefacts shall be passed back to VIL (this is not generically supported). Thus, we provide a simple extension mechanism for integrating custom instantiators with EASy-Producer.

In the first part of this section, we will introduce the basic instantiation concept of EASy-Producer to form a common understanding of how an instantiator works. In the second part, we will describe how to set-up a new plug-in project in Eclipse for implementing a custom instantiator. This also includes the specific configurations that have to be done to utilize the automated search and integration mechanism provided by Eclipse Dynamis Services. The third part will discuss the methods that are required when implementing a new instantiator. The focus of this part will be on how, when and why EASy-Producer invokes specific methods of an instantiator. In the fourth part, we will finally show how to integrate a new instantiator.

3.1.1. Instantiation Concept in EASy-Producer

In this section, we will introduce the basic instantiation concept in EASy-Producer in order to describe how the instantiators work. In the first part, we will have a black-box view on a generic instantiator for identifying the required input (prerequisites) for an instantiator. Please note that the generic instantiator is not related to any specific variability implementation technique (VIT). Thus, we can only give a very simple view on the instantiators in general. In the second part, we will relate the identified prerequisites in terms of giving a white-box view on the generic instantiator. However, the actual logic that defines how to process artefacts depends on the used VIT. Thus, this view is again simplified.

The concept of instantiators are closely related to the Variability Implementation Language (VIL) for specifying the individual build tasks of an instantiation process. From the perspective of VIL, instantiators are reusable, black-box components that may be called as part of a specific rule in an VIL build script or as part of an VIL template. Please note that we will only discuss those parts of VIL in this guide that are relevant to the actual implementation of an instantiators (and new artefact types in Section 3.2). For further details about the language concepts, the available types, and their application, please considere the VIL language specification (cf. Section 2.4).

An instantiator in EASy-Producer in general takes a set of possibly different input and produces a set of output. Typically, the input consists of a configuration based on an IVML variability model, generic artefacts (i.e. the artefacts of a software product line including variation points, etc.), and different variants that can be applied to the variation points of the generic artefacts. Please note that the presents as well as the location of generic artefacts, variation points and variants depends on the used VIT (we will detail this below). Based on this input, the instantiator produces an instantiator- or VIT-specific output, typically, a set of product-specific artefacts with resolved variability as illustrated by Figure 3.

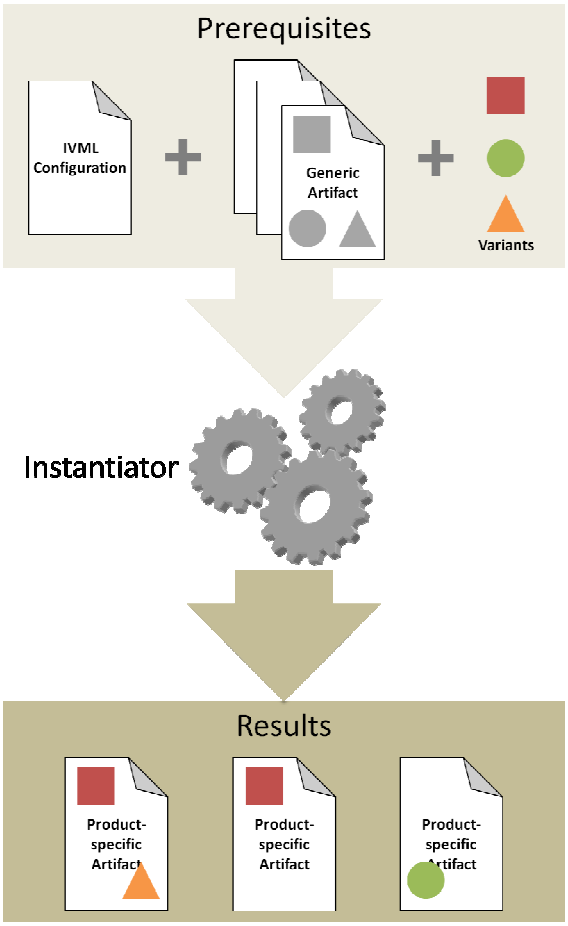

Figure 3: Black-box view of a generic instantiator (simplified)

Below, we will describe the required input (prerequisites) of an instantiator in detail:

- **IVML Configuration:** The IVML configuration is based on the previously defined variability model using the IVML modeling language (explicit prerequisite not mentioned in Figure 3). A configuration includes all variable-value pairs, which are valid with respect to the constraints defined in the model. The validity of such a configuration is automatically checked before the instantiation process. This prevents from calling the instantiator with an invalid configuration, which will yield corrupted product-specific artefacts. For instantiation, only configured and frozen variables can be considered.
- **Generic Artefacts:** The generic artefacts, i.e. of a software product line, include variation points (indicated by gray shapes in Figure 3) to which one or multiple variants (indicated as colored shapes in Figure 3) can be bound. However, the way of specifying such variation points (and the related variants) depends on the used VIT. For example, using preprocessing as a VIT, the variation points might be indicated by #if-statements in the generic artefacts. In some situations an instantiator may also generate artefacts from scratch, which does not require any generic artefacts as an input to the instantiator.
- **Variants:** The different variants that can be applied to a variation point of a generic artefact may be implemented independent from the generic artefacts. However, this also depends on the used VIT. In the example of preprocessing, the different variants will be part of the generic artefacts. The variants not selected as part of the product will then be

deleted by the preprocessor. In case of using aspect-orientation as VIT, the variants are implemented as independent aspects, which can be woven into the generic artefacts if they are selected as part of the product.

The relation between IVML configuration, generic artefacts, and variants is illustrated in Figure 4. The decision variables and their values will be passed in as a VIL configuration instance, which exactly defines the scope, while the files will be passed in as a VIL container of type FileArtifact (we will discuss this in detail in Section 3.1.3). The way of processing this information depends on the implemented instantiator logic. Figure 4 sketches two possible variants of such logic in pseudo-code:

- **Variant A:** In variant A the instantiator will process all files given by the VIL contanier, i.e. in terms of searching for variation points in each file (this is described as VIT-statement in Figure 4; some VITs may also introduce further concepts besides variation points that can be searched and processed by an instantiator). However, what to do if a certain variation point (or VIT-statement) is found heavily depends on the used VIT and the intention of the domain engineer who defines these variation points. Thus, we cannot give further information regarding the actual logic.
- **Variant B:** In variant B the instantiator will process all decision variables given by the VIL configuration, i.e. in terms of searching for a specific variable-value pair. However, what to do if a certain variable-value pair is found heavily depends on the used VIT and the relation between the specific decision variable and the artefacts. Thus, we cannot give further information regarding the actual logic.

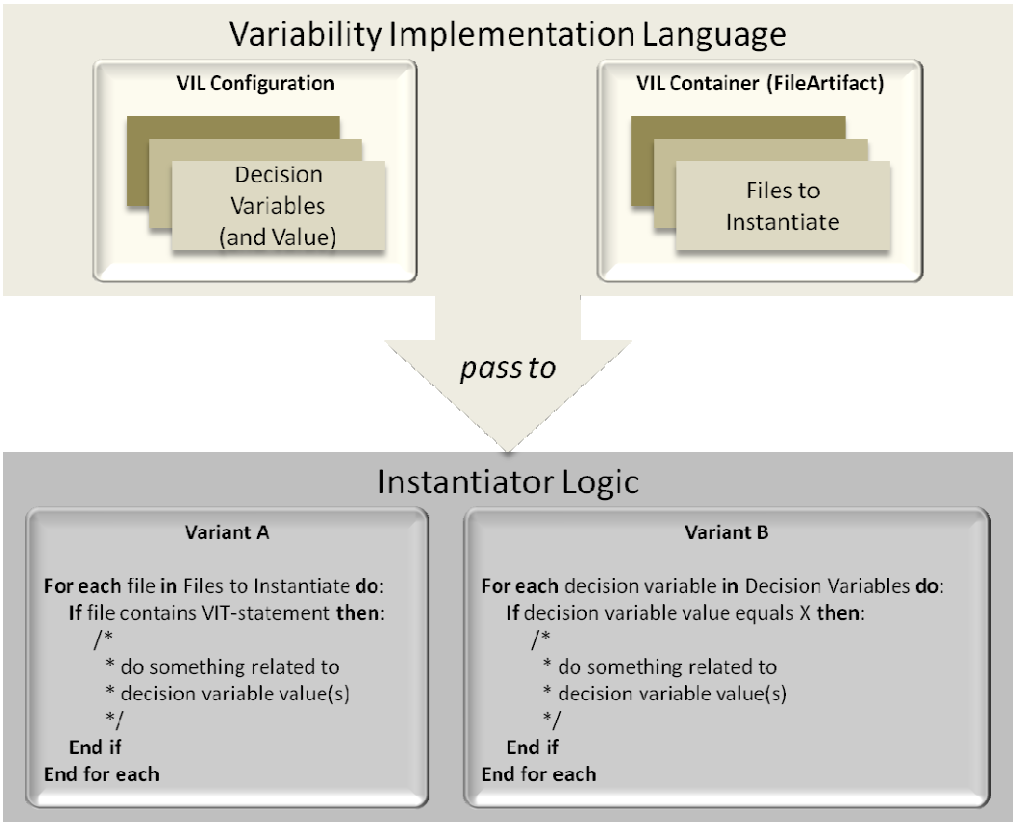

Figure 4: White-box view of a generic instantiator (simplified)

An instantiator may also provide further functionality, i.e. the generation of files based on the variable-value pairs (this may also exclude the selection of files to instantiate as the instantiation process will generate completely new files), the combination of other (non-source) artefacts like documentation, etc. However, this depends on the used VIT and the specific purpose an instantiator is designed for.

3.1.2. Eclipse Plug-in Project Creation and Configuration for New Instantiators

The first step towards a new instantiator is to create new Eclipse plug-in project: *File* → *New* → *Project…* . In the emerging wizard, open the category *Plug-in Development*, select *Plug-in Project*, and click the *Next* button. In the *New Plug-in Project* wizard, define a name for your project. We will use the following name throughout this section: *EASyExampleInstantiator*. Further, define the target platform with which the plug-in should run. In this case, the instantiator plug-in will run with a *standard OSGi framework*. Figure 5 shows how the first configuration page for the new plug-in project must look like.

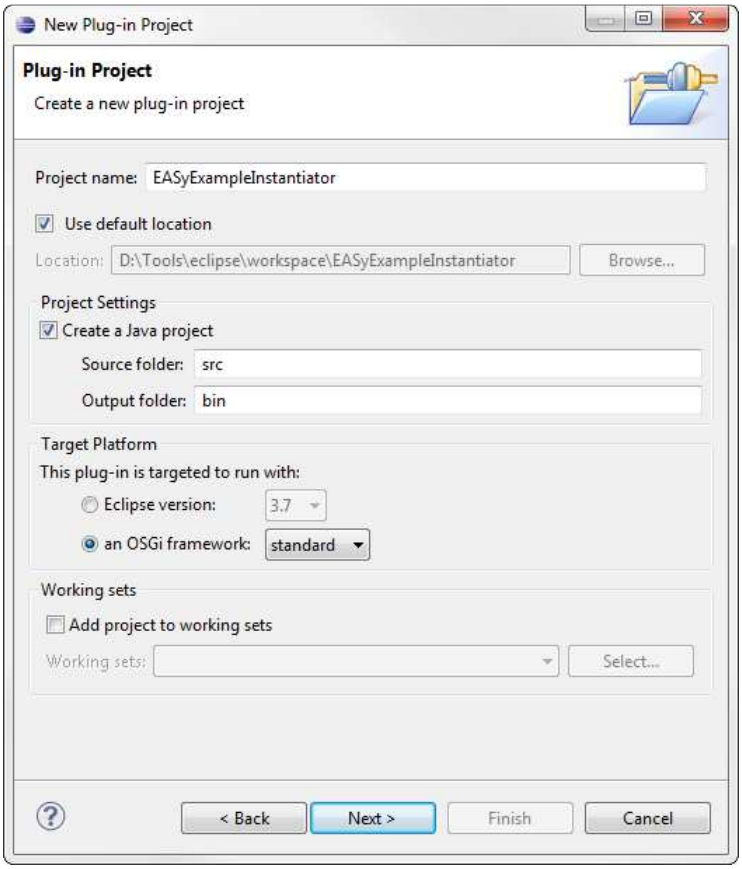

Figure 5: Configuration of a new Eclipse plug-in project for a new Instantiator

Click on the *Next* button and define the properties of your plug-in. We will use the following values for the required properties:

- *ID*: de.uni_hildesheim.sse.easy.instantiator.exampleInstantiator
- *Version*: 0.0.1
- *Name*: EASyExampleInstantiator
- *Provider*: University of Hildesheim – SSE

Leave all other properties and options as-is and finish the configuration by clicking the *Finish* button of the *New Plug-in Project* wizard. Please note that some of the following steps described in this section can also be done by using the wizard. However, we decided to do these steps manually to provide a more detailed explanation.

The plug-in manifest file will open by default. In the *Overview* tab check the *Activate this plug-in when one of its classes is loaded* checkbox and the *This plug-in is a singleton* checkbox. The first check will guarantee that the plug-in is activated when EASy-Producer loads one of its classes, while the second check is related to one of the concepts of EASy-Producer: each instantiator exists only once (only one instance) and can be accessed by any product line project. Thus, this check guarantees that the new instantiator will follow the concepts of EASy-Producer.

The next step is to define the dependencies of the new plug-in. Thus, open the plug-in manifest and select the *Dependencies* tab. On the left side click the *Add…* button in order to specify the following plug-ins:

- *org.eclipse.equinox.ds*: This plug-in simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies[6].
- *org.eclipse.core.runtime*: This plug-in provides support for the Eclipse runtime platform, core utility methods, and the extension registry[7]. The latter is important for the EASy-Producer extension mechanism.
- *de.uni-hildesheim.sse.easy.instantiatorCore*: This plug-in provides the core capabilities of the EASy-Producer instantiator concept. We will use parts of this plug-in in Section 3.1.3.

By default, Eclipse adds the package *org.osgi.framework* as *Imported Packages* because of the selected target platform in the *New Plug-in Project* wizard. However, this package is not required for the integration with EASy-Producer and, thus, can be removed. Select the package on the right side of the *Dependencies* tab and click the *Remove* button. Then, click the *Add…* button and select *org.osgi.service.component* as *Imported Packages*. This package provides support for service components and their interaction with the context in which they are executed[8]. The *Dependencies* tab should now look like the one illustrated by Figure 6.

[6] For more information visit: http://eclipse.org/equinox/
[7] For more information visit: Eclipse API – org.eclipse.core.runtime
[8] For more information visit: OSGi API – org.osgi.service.component

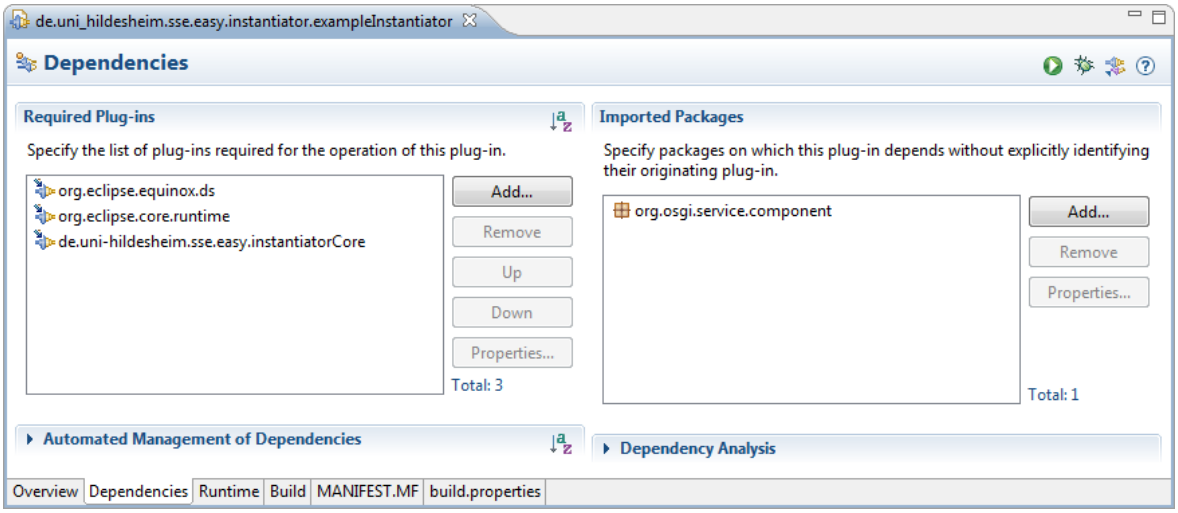

Figure 6: Definition of the required plug-ins for the new instantiator

In order to register the new plug-in to EASy-Producer, the service component has to be declared. Thus, switch to the *MANIFEST.MF* tab in the plug-in manifest and add the following *Service-Component* declaration:

```
Service-Component: OSGI-INF/instantiator.xml
```

This Service-Component declaration specifies the location where to find the information about the service component, which shall be integrated into EASy-Producer. The declared XML file will be defined in the next step. Figure 7 shows how the manifest file must look like.

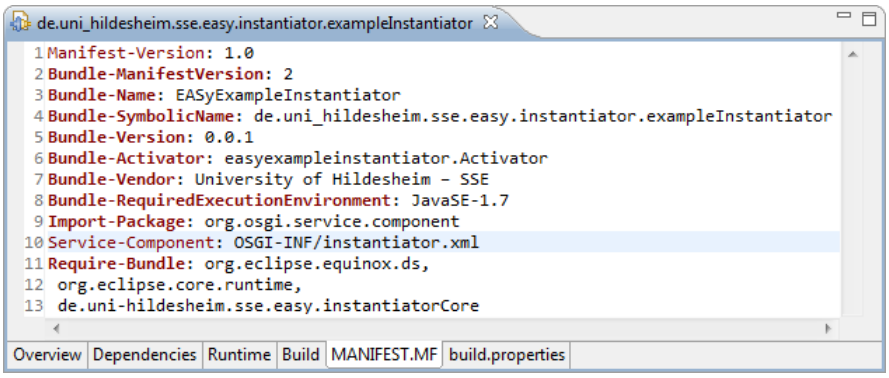

Figure 7: Declaration of the service-component for the new instantiator

The definition of the service component requires the creation of a new folder within the plug-in project. Right click on the plug-in project and select *New → Folder*. The name of the folder has to be *OSGI-INF*. Then, create a new XML file within this folder. Right click on the folder and select *New → Other...* . In the emerging wizard, open the category *XML*[9], select *XML File*, and click the *Next* button. Define the name of the file in accordance to the file declared in the manifest illustrated in Figure 7: *instantiator.xml*. Clicking the *Finish* button will open the XML editor. Switch to the source tab and edit the file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

[9] If the category *XML* does not exist, install XML support using *Help → Install New Software* or open the category *General*, select *File*, and define the name as well as the file-type manually.

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  immediate="true"
  name="EASy Example Instantiator">

  <implementation class="easyexampleinstantiator.ExampleEngine"/>

  <service>
    <provide interface="de.uni_hildesheim.sse.easy_producer.
      instantiator.InstantiatorEngine"/>
  </service>
</scr:component>
```

Figure 8 shows the final XML file. Please note that we used the names and package-structure of our example in Figure 8. Thus, with respect to different implementations the name of the service component in line 4 as well as the package and the class name of the implementation class element in line 6 (the class, which will implement the instantiator) have to be adapted. Please ignore the warning in line 6 as the class currently does not exist. This will be part of Section 3.1.3.

Figure 8: Definition of the service-component for the new instantiator

The previously defined XML file must be included in the binary build. Thus, open the manifest file again and switch to the *Build* tab. In the left lower part of this tab select the *OSGI-INF* folder to be included in the binary build.

The last step is the inclusion of external, third-party libraries – the actual instantiator. Please note that this step is only required if the main implementation of the instantiator or other required functionalities are implemented in another plug-in or library. In such a case, build the plug-in or the library first[10]. Then, right click on the current instantiator plug-in project, select *New* and *Folder*. The name of the new folder must be *lib*. Include all libraries in this folder that are required by the new instantiator. The folder and the required libraries have to be included in the *Classpath* of the new plug-in. Thus, open the plug-in manifest and switch to the *Runtime* tab. Add the libraries to the *Classpath* by clicking on the *Add…* button on the right side of the *Runtime* tab. Select all required libraries of the *lib* folder and click the *Ok* button. Switch to the *Build* tab of the plug-in manifest and select the *lib* folder to be part of the *Binary Build* in the left lower part of this tab. Figure 9 and Figure 10 show the result in the context of our example. Figure 9 shows the included library de.uni_hildesheim.sse_0.0.1.jar, which provides the main functionality of our prototypical instantiator and, thus, has to be available at runtime. Figure 10 illustrates the build configuration in which the library (highlighted) is selected as part of the *Binary Build*.

[10] If you do not know how to build a plug-in, please consider Section 3.1.3.

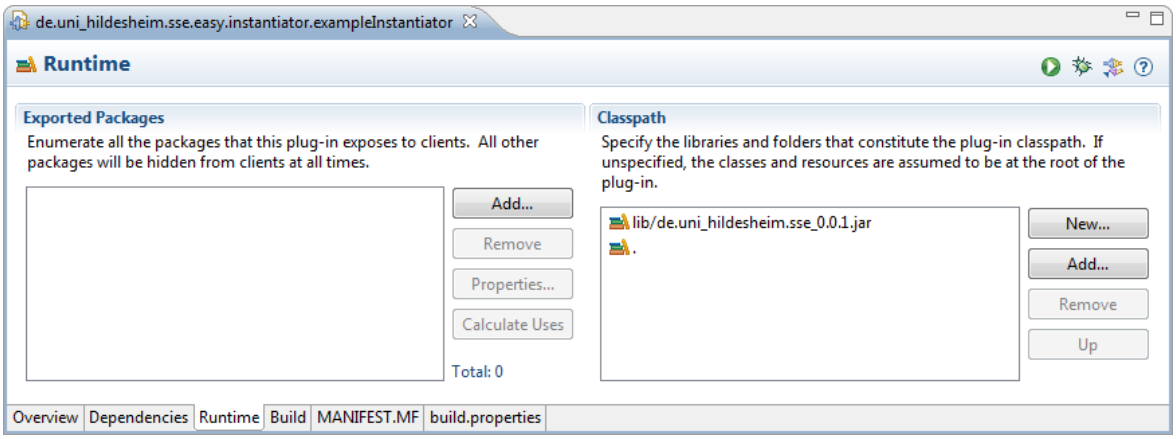

Figure 9: Classpath specification of external, required libraries

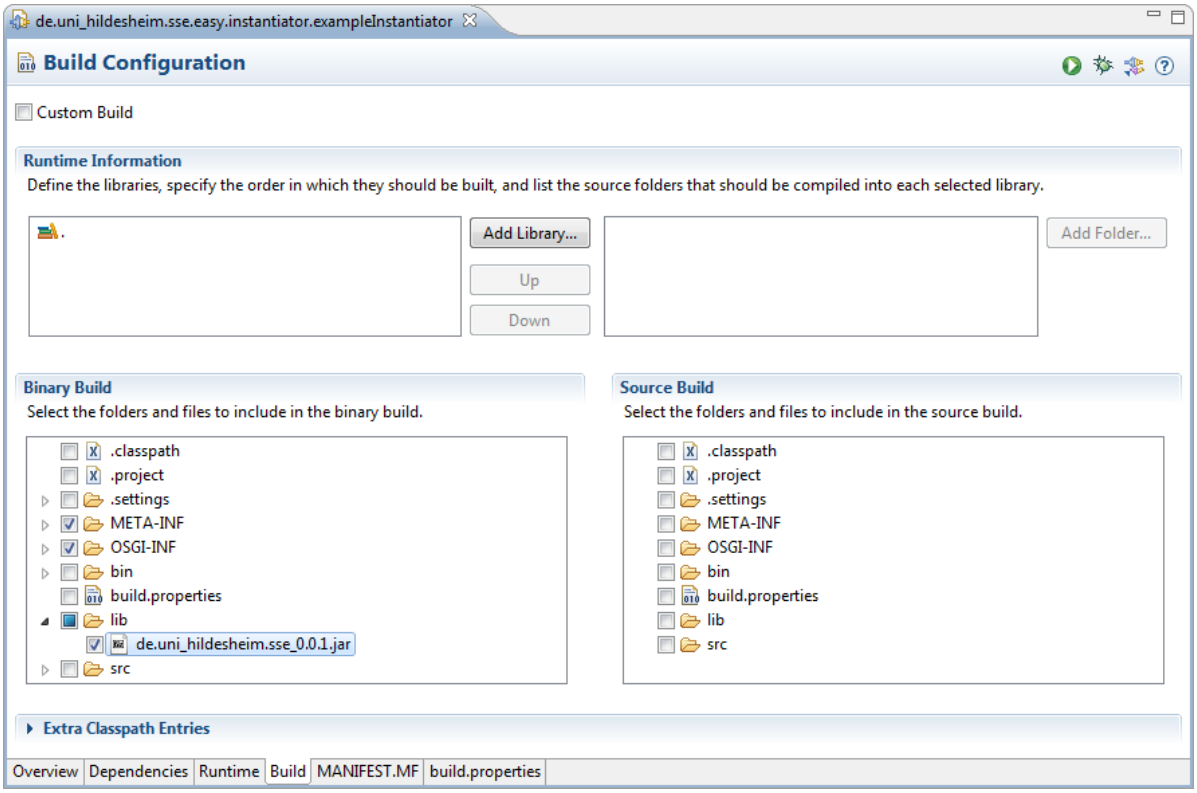

Figure 10: Binary Build selection of external, required libraries

Finally, the plug-in project is set up, configured and ready to use. In the next section, we will further develop this plug-in by implementing instantiator-specific functionality based on the results of this section.

3.1.3. Instantiator Implementation

In the previous section, we set up the Eclipse plug-in project for implementing a new instantiator for EASy-Producer. In this section, we will describe how to implement the (basic) functionalities of an instantiator. However, as each instantiator provides its individual capabilities and is used for different purposes, this description will only include the basic functionalities that are common to each instantiator.

The first step is to create a new Java class file. Right click on the package that was defined as the implementation class package in Section 3.1.2 and select *New → Class*. In the emerging *Java Class* wizard, define the name of the new class in accordance to the name of the implementation class (cf. Section 3.1.2). In our example, we use the name *ExampleEngine*. Leave all other options as-is.

Each instantiator must implement the *IVilType* interface, must be annotated with the annotation Instantiator, and must implement at least one static method, which typically has the same name as the instantiator (because the name of the method will as well identify the instantiator call in VIL). This enables the integration of the new instantiator as part of the VIL language. We will describe this intergration in detail in Section 3.1.4, while details about VIL types in gereral, the annotation, and, in particular, the *IVilType* interface can be obtained in the VIL language specification (cf. Section 2.4).

Thus, the next step is to edit the new class file as follows (please note that we use the packages and class names of our example):

```
package easyexampleinstantiator;

import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel
        .FileArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
        .ArtifactException;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
        .Collection;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
        .IVilType;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
        .Instantiator;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.Set;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
        .configuration.Configuration;

@Instantiator("exampleEngine")
public class ExampleEngine implements IVilType {

 protected void activate(ComponentContext context) {
       try {
             TypeRegistry.registerType(ExampleEngine.class);
       } catch (VilException e) {
             e.printStackTrace();
       }
 }

 protected void deactivate(ComponentContext context) {

 }

 public static Set<FileArtifact> exampleEngine(Collection<FileArtifact>
       templates, Configuration config) throws ArtifactException {
       // Implementation of the actual instantiation
 }

 // ...
}
```

We will now discuss each of these methods in detail:

- ***activate*:** This method is used to activate the instantiator plug-in. In this case, we will register the new type in the type registry. This will include the type in the artefact model, ready for use in the VIL build language or the template language.
- ***deactivate*:** This method is used to deactivate the instantiator plug-in. However, in this situations we do not need to unregister the type again as this would yield errors in the VIL build script or template as the type would be unknown.

The actual implementation above is rather simple. The single static method represents the entry-point of the instantiator when it is called as part of a VIL build execution (mandatory). Here, the name of the method *exampleEngine* will be used in the VIL language to call this instantiator, including the defined parameters. In our example, this method requires the follwing parameters:

- *templates*: This collection includes a set of *FileArtifacts*, which represent (real) generic artefacts, for example, of a specific software product line. The *Collection* type as well as the *FileArtifact* type are defined in the **Artifact Model** of VIL (see VIL Language Specification for details on the VIL artefact model). This set of artefacts will be processed by the instantor depending on the actual logic.
- *config*: The current configuration based on the IVML variability model of the respective product line. The *Configuration* type is again part of the VIL artefact model. The configuration provides access to the current variables and their values to determine which artefacts have to be instantiated in which way. However, this is defined in the actual implementation of the instantiator.

Please note that the example above only provides a prototypical implementation of an instantiator. The types used in the actual implementation depend on the logic of the instantiator and its purpose. The available types in turn depend on the VIL type system.

The last step is to build the plug-in. Open the plug-in manifest file and click on the *Export deployable plug-ins and fragments* button in the upper right corner. In the emerging wizard select the current plug-in project, specify the desired destination[13], and click on the *Finish* button.

Finally, the plug-in and, thus, the instantiator is implemented, build, and ready for use. In the next section, we will describe how to integrate a new instantiator in EASy-Producer. We will also have a quick look on how to use it. However, for detailed information on how to use an instantiator, please consider the EASy-Producer User Guide.

3.1.4. Instantiator Integration

In the previous section, we implemented the (basic) functionalities of a new instantiator. Further, we build a deployable plug-in, which we will use in this section for integrating the new instantiator within an EASy-Producer installation.

The first and only step is to copy the previously build instantiator plug-in into the *dropins* folder of the Eclipse application in which EASy-Producer installed. Start the Eclipse application and create a new product line project: *File* → *New* → *Other…* → *EASy-Producer* → *New EASy-Producer Project*.

[13] The destination is up to the developer. However, we recommend using a location, which is easy to find as we will need the location for integrating the new instantiator in Section 3.1.4.

The name of the new project is up to the developer. If a product line project is available in the workspace, open, for example, the **VIL Build Language Editor** by double clicking the VIL file in the **EASy-folder**. Calling the new instantiator as part of a build task can be done by simple typing the name of the method defined in Section 3.1.3 and passing the required parameters. Figure 12 shows the call of our example instantiator.

PL_Content_Sharing_0.vil

```
1  vilScript PL_Content_Sharing (Project source, Configuration conf, Project target) {
2
3      version v0;
4
5      main(Project source, Configuration conf, Project target) = : {
6          exampleEngine(source.selectAllFiles(), conf);
7      }
8
9  }
```

Figure 12: Using the new instantiator in a VIL build script.

3.2. Implementing a New VIL Artefact Type

The Variability Implementation Language (VIL) is a textual language for the flexible specification of the instantiation process of a software product line or any other software project that includes variabilities. Actually, VIL is not a single language. It consists of four main parts, namely the artefact model, the VIL template language, blackbox instantiators, and the VIL build language. In this section, we will focus on the artefact model and the extension of this model by new artefact types. In the first part, we will briefly introduce the VIL artefact model and discuss the basic concept regarding the extension capabilities. In the second part, we will describe the extension of the model by an example artefact in a step-wise manner.

3.2.1. The VIL Artefact Model in EASy-Producer

The artefact model defines the individual capabilities of various types of assets used in variability instantiation, such as Java source code, Java byte code, XML files but also components (for runtime variabilities), or elements of the file system such as files or folders. Production strategies are operations on the types of the input and output artefacts using the capabilities of the assets for specifying the instantiation. Thus, the available artefact types will be used in the VIL build language and the VIL template language to enable the instantiation of variable artefacts of the respective type. More details on the artefact model and VIL in general can be found in the VIL language specification (cf. Section 2.4).

The classes in the artefact model can be understood as meta-classes of artefacts. Instances of these classes represent real artefacts. Artefacts are VIL types in order to be available in the VIL editors. Currently, there are five fundamental types:

- ***Path expressions*** for denoting file system and language-specific paths.
- ***Simple artefacts***, which cannot be decomposed. Typically, generic folders and simple generic components shall be represented as simple artefacts. Some of those artefacts act as default representation through the ArtefactFactory, i.e., any real artefact which is not specified by a more specific artefact class is represented by those artefact types.

- ***Fragment artefacts***, representing decomposed artefact fragments such as a Java method or a SQL statement.
- ***Composite artefacts***, representing decomposable artefacts consisting of fragments. In case of resolution conflicts, composite artefacts have more priority than simple artefacts, e.g., if there is a simple artefact and a composite artefact representation of Java source classes, the composite will be taken. However, if there are resolution conflicts in the same type of artefacts, e.g., multiple composite representations, then the first one loaded by Java will take precedence. In order to implement a decomposable artefact, also an instance of the **IArtifactCreator** must be implemented. This describes creator instances which know how to translate real world objects into artifact instances. We will illsutrate the usage of this interface in the implementation of our example in Section .3.2.3
- The types in de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes are most basic and enable building a bridge to the variability model with own VIL-specific operations.

Instances of all artefact types can be obtained from the **ArtefactFactory**. This is in particular true for instances of the **ArtefactModel** which provides an environment for instantiating artefacts, i.e., it manages created artefacts. While the **ArtefactFactory** may be used standalone, the correct internal listener registration is done by **ArtefactModel** so that model and artefacts are informed about changes and can be kept up to date, i.e., artefact instances shall be created using methods of **ArtefactModel**.

Subclassing these artefact types (and registering them with the artefact factory through the Eclipse DS mechanism) transparently leads to more specific artefact types with more specific operations. Please note that even the simple names of Vil types and artefacts shall be unique (unless they shall override existing implementations) due to the transparent embedding into the VIL languages. Types must be registered in **TypeRegistry**.

All operations marked by the annotation **Invisible** will not be available through the VIL languages. However, the (semantics of the) Invisible annotation may be inherited if required. By convention, collections are returned in terms of type-parameterized sets or sequences. However, an artefact method returning a collection must be annotated by **OperationMeta.returnGenerics()** in order to defined the actual types used in the collection (this is not available via Java mechanisms). Further, operations and classes may be marked by the following annotations:

- ***Conversion*** to indicate type conversion operations considered for automatic type conversion when calling methods from a VIL expression. These methods must be static, take one parameter of the source type and return the target type.
- ***OperationMeta*** for renaming operations (for operator implementations), determining their operator type or, as mentioned above, making the type parameters of a generic return type explicit. Basically, all three information types are optional.
- ***ClassMeta*** for renaming the annotated class, i.e., hiding the Java implementation name.

Collections may define generic iterator operations such as checking a condition or applying a transformation expression to each element. Therefore, a non-static operation on a collection receiving (at the moment exactly) one **ExpressionEvaluator** instance as parameter (possibly more

parameters) will be considered by VIL as an iterator operation. The **ExpressionEvaluator** will carry an iterator variable of the first parameter (element type) of the collection as well as an expression parameterized over that variable (i.e., it uses the [undbound] variable). The job of the respective collection operation is to apply the expression to each element in the collection, i.e., to bind the variable to each collection element (via the runtime variable of the temporarily attached **EvaluationVisitor** in the **ExpressionEvaluator**), to call the respective evaluation operation of the ExpressionEvaluator and to handle the returned evaluation result appropriately.

Artefact or instantiator operations may cause VIL rules and templates to fail if they return a non-true result, i.e., an empty collection or null. However, in order to state explicitly that an operation cannot be executed, an operation shall throw an **ArtefactException**.

Basically, artefact or instantiator operations are identified by their name, the number, sequence and type of their parameter. However, some operations such as template processors may require an unlimited number of not previously defined parameters. In this case, VIL allows to pass in named parameters. In the respective artefact or instantiator operations, named parameters are represented by a **Map** as last parameter which receives the names and the actual values of given named VIL parameters. The interpretation of named parameters belongs to the respective method.

3.2.2. Eclipse Plug-in Project Creation and Configuration for New Artefacts

The set up of an Eclipse plug-in project for a new VIL artefact type is quite similar to the set up for a new instantiation described in Section 3.1.2. Below, we will decribe the only changes with respect to the instantiator set up:

- ***Project name:*** We will use EASyExampleArtifact as the name for the project throughout this sections.
- **OSGI information:** The XML-file for the definition of the service component will change in terms of the name (artifact.xml) and the content as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
   immediate="true"
   name="EASy Example Artifact">

   <implementation class="easyexampleartifact.ExampleArtifact"/>

   <service>
      <provide interface="de.uni_hildesheim.sse.easy_producer.
         instantiator.model.vilTypes.IVilType"/>
   </service>
</scr:component>
```

3.2.3. Artefact Implementation

The first step is to create a new Java class file. In our example, we use the name *ExampleArtifact*. Each new artefact has to implement the *IVilType* interface and may extend one of the base VIL

types introduced in Section 3.2.1. We will extend the **FileArtifact** in this example. Thus, the content[14] of the new class file looks like this:

```
package easyexampleartifact;

import java.io.File;

import org.osgi.service.component.ComponentContext;

import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.ArtifactCreator;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.ArtifactModel;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.FileArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.FragmentArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.IArtifactVisitor;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.representation.
       Binary;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.representation.Text;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.ArtifactException;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.IVilType;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.Set;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.TypeRegistry;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.VilException;

@ArtifactCreator(ExampleFileArtifactCreator.class)
public class ExampleArtifact extends FileArtifact implements IVilType {

	public ExampleArtifact(File file, ArtifactModel model) {
		super(file, model);
	}

	protected void activate(ComponentContext context) {
		try {
			TypeRegistry.registerType(ExampleArtifact.class);
		} catch (VilException e) {
			e.printStackTrace();
		}
	}

	protected void deactivate(ComponentContext context) {

	}

	@Override
	public void delete() throws ArtifactException {
		// Here goes the implementation
	}

	@Override
	public String getName() throws ArtifactException {
		// Here goes the implementation
	}

	@Override
	public void rename(String name) throws ArtifactException {
		// Here goes the implementation
	}
```

[14] Please note that we cannot discuss all methods in detail due to the complexity of the artefact model and the available types and methods.

```
	@Override
	public void accept(IArtifactVisitor visitor) {
		// Here goes the implementation
	}

	@Override
	public boolean isUptodate(long timestamp) {
		// Here goes the implementation
	}

	@Override
	public boolean exists() {
		// Here goes the implementation
	}

	@Override
	public void artifactChanged() throws ArtifactException {
		// Here goes the implementation
	}

	@Override
	public Set<? extends FragmentArtifact> selectAll() {
		// Here goes the implementation
	}

	@Override
	protected Text createText() throws ArtifactException {
		// Here goes the implementation
	}

	@Override
	protected Binary createBinary() throws ArtifactException {
		// Here goes the implementation
	}

	@Override
	public void store() throws ArtifactException {
		// Here goes the implementation
	}
}
```

We will now discuss each of these methods in detail:

- **Constructor:** The constructor of this class requires a (real) artefact in terms of a file, which will be represented by this artefact type, and the corresponding artefact model instance this artefact belongs to. These parameters are passed to the super-class, the **FileArtifact**.
- ***activate*:** This method is used to activate the instantiator plug-in. In this case, we will register the new type in the type registry. This will include the type in the artefact model, ready for use in the VIL build language or the template language.
- ***deactivate*:** This method is used to deactivate the instantiator plug-in. However, in this situations we do not need to unregister the type again as this would yield errors in the VIL build script or template as the type would be unknown.
- ***delete*:** This method deletes the current instance of this artefact Including its underlying real-world object, e.g., this operation may delete an entire file.
- ***getName*:** This method returns the name of the current instance of this artefact.

- ***rename*:** This method renames the current instance of this artefact and its underlying real-world object.
- ***accept*:** This method visits the current instance of this artefact (and dependent on the visitor also contained artifacts and fragments) using the given visitor.
- ***isUptodate:*** This method returns whether the current instance of this artefact is up-to-date (with respect to the given timestamp) and whether it shall be considered for recreation in preconditions of VIL build language rules.
- ***exists:*** This method returns whether the current instance of this artefact exists. Also this method is considered by the VIL build language.
- ***artifactChanged:*** This method is called when the current instance of this artefact was changed, e.g., to trigger a reanalysis of substructures. This may be caused by one of the alternative basic representations such as text or binary (see below).
- **selectAll:** This method returns all artefacts of the current instance of this composite artefact is composed of.
- **createText:** This method actually creates a text representation of the current instance of this artefact. The binary representation enables to modify the entire artifact from a textual point of view, i.e., using text manipulation operations. Please note that the **getText**-method of the super-class **CompositeArtifact** calls this method and registers the listeners appropriately.
- **createBinary:** This method actually creates a binary representation of the current instance of this artefact. The binary representation enables to modify the entire artifact from a binary point of view, i.e., in terms of individual bytes. Please note that the **getBinary()**-method of the super-class **CompositeArtifact** calls this method and registers the listeners appropriately.
- **store:** This method stores changes to the artefact. Typically, this is done by saving the changes of the file contents to the real-world file artefact.

The new artefact type is derived from the **FileArtifact** type of the VIL artefact model, which is derived from the **CompositeArtifact** introduced in Section 3.2.1. This type of artefact also requires the implementation of an instance of the **IArtifactCreator** interface to relate real-world artefacts to the new VIL type (indicated by the annotation of this class-implementation above). Implementations of this artefact must fulfill the following contract:

- The method **handlesArtifact(Class, Object)** of the **ArtifactCreator** is called to figure out whether a creator (**Class**) is able to handle a certain artifact (**Object**) under given class-based restrictions. Typically, more specific creators are asked later than more generic ones, but more specific creators (according to inheritance relationships) are considered first for creation. An implementation which answers <code>true</code> must be able to create the queried artifact.
- The method **createArtifactInstance(Object)** of the **ArtifactCreator** actually creates an instance for the previously queried object. However, no information shall be stored nor there is a guarantee that this method will be called (dependent on the other registered creators). As stated above, if **handlesArtifact(Class, Object)** answers with *true*, **createArtifactInstance(Object)** must be able to perform the creation for the given object.

In our example, we will define a new class for the implementation of an artefact creator, named ExampleFileArtifactCreator. The implementation of this class is given below:

```
package easyexampleartifact;

import java.io.File;

import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.ArtifactModel;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.
       DefaultFileArtifactCreator;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.FileArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.IArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.ArtifactException;

public class ExampleFileArtifactCreator extends DefaultFileArtifactCreator {

       @Override
       protected boolean handlesFileImpl(File file) {
             return checkSuffix(file, ".example");
       }

       @Override
       public FileArtifact createArtifactInstance(Object real, ArtifactModel model)
         throws ArtifactException {
             return new ExampleArtifact((File) real, model);
       }

       @Override
       public Class<? extends IArtifact> getArtifactClass() {
             return ExampleArtifact.class;
       }
}
```

We will now discuss each of these methods in detail:

- **handlesFileImpl:** This method may specify additional properties of a file that must be fullfilled in order to define the file as representable by this artefact type (the **ExampleArtifact** in this case). It is already guaranteed that the passed file is a file and not a directory. In our example, we will check whether the suffix of the given file matches "*.example*".
- **createArtifactInstance:** This method creates a new instance of the artefact type based on the passed (real) artefact.
- **getArtifactClass:** This method return the class that implements the artefact.

The last step is to build the plug-in. Open the plug-in manifest file and click on the *Export deployable plug-ins and fragments* button in the upper right corner. In the emerging wizard select the current plug-in project, specify the desired destination[15], and click on the *Finish* button.

Finally, the plug-in and, thus, the new artefact is implemented, build, and ready for use. The integration is the same as in the instantiator case described in Section 3.1.4 (copy the final plug-in into the dropins-folder). The next time Eclipse will be started the new artefct type can be used in the VIL build language and in the VIL template language.

[15] The destination is up to the developer. However, we recommend using a location, which is easy to find as we will need the location for integrating the new instantiator in Section 3.1.4.

3.3. Implementing a New Reasoner

The IVML language provides highly expressive modelling elements and concepts for the definition of variability models. Thus, checking whether a specific (product) configuration is valid is a challenging task. In EASy-Producer, we use so-called reasoners to perform the task of model and configuration checking and validation. A reasoner is typically a third-party tool, which is designed to solve logical and combinatorial problems, checking specific value combinations of related modelling elements, etc. Similar to the instantiators in EASy-Producer, we provide a simple extension mechanism for integrating custom reasoners with the tool.

In the following sections, we will describe the set-up a new plug-in project in Eclipse for implementing a custom reasoner. This also includes the specific configurations that have to be done to utilize the automated search and integration mechanism provided by EASy-Producer. Further, we will discuss the methods that are required when implementing a new reasoner.

3.3.1. Eclipse Plug-in Project Creation and Configuration for New Reasoners

The first steps of the creation of a new Eclipse plug-in for the implementation of a new reasoner are quite similar to the creation of a new instantiator plug-in (cf. Section 3.1.2). However, the first change is in the name of the new project. We will use *EASyExampleReasoner* throughout this section as the name and define, again, the *standard OSGI framework* as the target platform.

Further changes occur in the definition of the plug-in properties. We will use the following values:

- *ID*: de.uni_hildesheim.sse.easy.reasoner.exampleReasoner
- *Version*: 0.0.1
- *Name*: EASyExampleReasoner
- *Provider*: University of Hildesheim – SSE

The plug-in manifest file will open by default after clicking the *Finish* button. In the *Overview* tab check the *Activate this plug-in when one of its classes is loaded* checkbox and the *This plug-in is a singleton* checkbox.

The next step is to define the dependencies of the new plug-in. Thus, open the plug-in manifest and select the *Dependencies* tab. On the left side click the *Add…* button in order to specify the following plug-ins:

- *org.eclipse.equinox.ds* (described in Section 3.1.2)
- *org.eclipse.core.runtime* (described in Section 3.1.2)
- *ReasonerCore*: This plug-in provides the core capabilities of the EASy-Producer reasoning concept. We will use parts of this plug-in in Section 3.3.2.
- *de*.uni_hildesheim.sse.varModel: This plug-in provides access to the underlying variability object model of EASy-Producer. This provides, for example, the access to the current configuration of the variability model, the included decision variables and constraints, etc. We will use parts of this plug-in in Section 3.3.2.

By default, Eclipse adds the package *org.osgi.framework* as *Imported Packages* because of the selected target platform in the *New Plug-in Project* wizard. However, this package is not required for the integration with EASy-Producer and, thus, can be removed. Select the package on the

right side of the *Dependencies* tab and click the *Remove* button. Then, click the *Add…* button and select *org.osgi.service.component* as *Imported Packages*. This package provides support for service components and their interaction with the context in which they are executed.

In order to register the new plug-in to EASy-Producer, the service component has to be declared. Thus, switch to the *MANIFEST.MF* tab in the plug-in manifest and add the following *Service-Component* declaration:

```
Service-Component: OSGI-INF/reasoner.xml
```

The definition of the service component requires the creation of a new folder within the plug-in project. Right click on the plug-in project and select *New → Folder*. The name of the folder has to be *OSGI-INF*. Then, create a new XML file within this folder. Right click on the folder and select *New → Other…* . In the emerging wizard, open the category *XML*[16], select *XML File*, and click the *Next* button. Define the name of the file in accordance to the file declared in the manifest: *reasoner.xml*. Clicking the *Finish* button will open the XML editor. Switch to the source tab and edit the file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
   immediate="true"
   name="EASy Example Reasoner">

   <implementation class="easyexamplereasoner.ExampleReasoner"/>

   <service>
      <provide interface="de.uni_hildesheim.sse.reasoning.core.
         reasoner.IReasoner"/>
   </service>
</scr:component>
```

The previously defined XML file must be included in the binary build. Thus, open the manifest file again and switch to the *Build* tab. In the left lower part of this tab select the *OSGI-INF* folder to be included in the binary build.

In order to include external, third-party libraries, follow the last step decribed in Section 3.1.2.

Finally, the plug-in project is set up, configured and ready to use. In the next section, we will further develop this plug-in by implementing the required classes and methods based on the results of this section.

3.3.2. Reasoner Implementation

In the previous section, we set up the Eclipse plug-in project for implementing a new reasoner for EASy-Producer. In this section, we will first describe the different classes that are required to register the new reasoner in EASy-Producer and, second, describe how to implement the required (basic) methods of a reasoner. However, as each reasoner provides its individual capabilities, this description will only include the basic functionalities that are common to each reasoner.

[16] If the category *XML* does not exist, install XML support using *Help → Install New Software* or open the category *General*, select *File*, and define the name as well as the file-type manually.

In contrast to the instantiator implementation, the implementation of a reasoner requires a two Java classes. Below, we will describe each required class in detail. Please note that the names of the classes are due to the name of this example.

- *ExampleReasonerDescriptor.java*: This class includes the following attributes of a reasoner: the descriptive name, the version, the license, license restrictions, and a download source (if this is a third-party reasoner). While the name is a mandatory attribute, all other attributes are optional. In this example, the reasoner descriptor looks like this:

```
package easyexamplereasoner;

import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasonerDescriptor;

public class ExampleReasonerDescriptor extends ReasonerDescriptor {

	static final String NAME = "Example Reasoner";

	static final String VERSION = "0.1";

	private static final String LICENSE = "<Licences Agreement>";

	public ExampleReasonerDescriptor() {
		super(NAME, VERSION, LICENSE, null, null);
	}

	@Override
	public boolean isReadyForUse() {
		return true;
	}
}
```

- *ExampleReasoner.java*: This class provides the actual implementation of the reasoner. In this example, the reasoner implementation looks like this:

```
package easyexamplereasoner;

import java.net.URI;
import java.util.List;

import org.osgi.service.component.ComponentContext;

import de.uni_hildesheim.sse.model.progress.ProgressObserver;
import de.uni_hildesheim.sse.model.varModel.Constraint;
import de.uni_hildesheim.sse.model.varModel.Project;
import de.uni_hildesheim.sse.reasoning.core.frontend.ReasonerFrontend;
import de.uni_hildesheim.sse.reasoning.core.reasoner.EvaluationResult;
import de.uni_hildesheim.sse.reasoning.core.reasoner.IReasoner;
import de.uni_hildesheim.sse.reasoning.core.reasoner.IReasonerMessage;
import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasonerConfiguration;
import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasonerDescriptor;
import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasoningResult;

public class ExampleReasoner implements IReasoner {

	private static final ReasonerDescriptor DESCRIPTOR = new
ExampleReasonerDescriptor();
```

```
    protected void activate(ComponentContext context) {
        ReasonerFrontend.getInstance().getRegistry().register(this);
    }

    protected void deactivate(ComponentContext context) {
        ReasonerFrontend.getInstance().getRegistry().unregister(this);
    }

      @Override
      public ReasonerDescriptor getDescriptor() {
            return DESCRIPTOR;
      }

      @Override
      public ReasoningResult upgrade(URI url, ProgressObserver observer) {
            // Here goes the implementation
      }

      @Override
      public ReasoningResult isConsistent(Project project,
                  ReasonerConfiguration reasonerConfiguration,
                  ProgressObserver observer) {
            // Here goes the implementation
      }

      @Override
      public void notify(IReasonerMessage message) {
            // Here goes the implementation
      }

      @Override
      public ReasoningResult check(Project project,
                  de.uni_hildesheim.sse.model.confModel.Configuration cfg,
                  ReasonerConfiguration reasonerConfiguration,
                  ProgressObserver observer) {
            // Here goes the implementation
      }

      @Override
      public ReasoningResult propagate(Project project,
                  de.uni_hildesheim.sse.model.confModel.Configuration cfg,
                  ReasonerConfiguration reasonerConfiguration,
                  ProgressObserver observer) {
            // Here goes the implementation
      }

      @Override
      public EvaluationResult evaluate(Project project,
                  de.uni_hildesheim.sse.model.confModel.Configuration cfg,
                  List<Constraint> constraints,
                  ReasonerConfiguration reasonerConfiguration,
                  ProgressObserver observer) {
            // Here goes the implementation
      }
}
```

We will now discuss each of these methods in detail:

- ***activate*:** This method is used to activate the reasoner plug-in. We recommend not changing this method in order to guarantee that EASy-Producer activates the reasoner properly.
- ***deactivate*:** This method is used to deactivate the reasoner plug-in. We recommend not changing this method in order to guarantee that EASy-Producer deactivates the reasoner properly.
- ***getDescriptor*:** This method returns the descriptor of this reasoner define above stating common information about this reasoner.
- ***upgrade:*** This method updates the installation of this reasoner, e.g., in order to obtain a licensed reasoner version if a third-party reasoner is used.
- ***isConsistent:*** This method is invokes by EASy-Producer if a given variability model should be checked for satisfiability. However, the actual implementation of this method depends on the reasoner.
- ***notify:*** This method is called when a reasoner message is issued.
- ***check:*** This method checks the configuration according to the given project structure.
- ***propagate:*** This method checks the configuration according to the given model and propagates values, if possible. The concept of value propagation defines the automatic assignment of currently unassigned decision variables of the configuration. This automation requires the assignment of a subset of the available decision variables and the relation of these variables to the unassigned variables in terms of constraints in the variability model.
- ***evaluate:*** This method evaluates a given list of constraints (in the sense of boolean conditions) which are related to and valid in the context of the given project and configuration.

The last step is to build the plug-in. Open the plug-in manifest file and click on the *Export deployable plug-ins and fragments* button in the upper right corner. In the emerging wizard select the current plug-in project, specify the desired destination[17], and click on the *Finish* button.

Finally, the plug-in and, thus, the reasoner is implemented, build, and ready for use. In the next section, we will describe how to integrate a new reasoner in EASy-Producer.

3.3.3. Reasoner Integration

In the previous section, we implemented the (basic) functionalities of a new reasoner. Further, we build a deployable plug-in, which we will use in this section for integrating the new reasoner in an EASy-Producer installation.

The first and only step is to copy the previously build reasoner plug-in into the *dropins* folder of the Eclipse application in which EASy-Producer installed. Start the Eclipse application and open the preference page of EASy-Producer: *Window* → *Preferences* → *EASy-Producer*. Expand the EASy-Producer category and select *Reasoners*. The new reasoner will be availabe as shown in Figure 13.

[17] The destination is up to the developer. However, we recommend using a location, which is easy to find as we will need the location for integrating the new reasoner in Section 3.3.3.

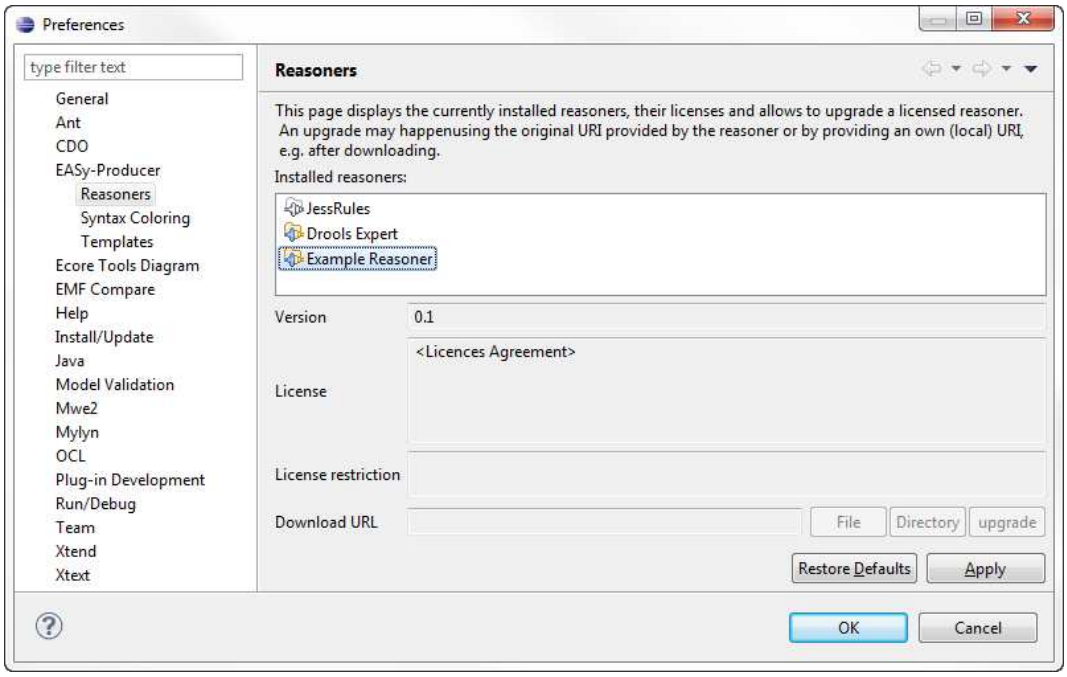

Figure 13: Reasoner preferences page